Getting liberated from the free list

O. Danvy

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.(1) 39 63 55 11

Rapports de Recherche

N° 735

GETTING LIBERATED FROM THE FREE LIST

Olivier DANVY

OCTOBRE 1987

Getting Liberated from the Free List

"Comment se libérer de la liste libre".

Olivier DANVY

Institute of Datalogy - University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, DENMARK
uucp: ...!mcvax!diku!danvy

Abstract

This paper describes why a free-storage list is an useless middle result. It is illustrated with the well-known mark and sweep garbage collection algorithm, and this improves a memory management in proportion of the free-list cost.

A free list is not for free: it is an intermediate and ephemeral structure built by the collector and destroyed by the mutator. It originates with LISP 1 and LISP 1.5, where a cons-cell was marked with its sign bit. More independent marks allow: (1) to concentrate unmarking just before the marking phase to start with an uniformly unmarked memory, since this is the only reason for resetting marks; (2) to get liberated from the free list by combining the sweep phase with the allocation. This leads to a lazy sweeping garbage collector. This paper presents it and justifies its correctness, extends it to freeing storage on-the-fly and concludes on its efficiency and parallelization.

Keywords: garbage collection, lazy sweeping, LISP, mark and sweep.

Résumé

Cet article décrit pourquoi une liste dite libre de blocs mémoriels prêts à être alloués est un résultat intermédiaire inutile. Ce point est illustré avec l'algorithme bien connu de glanage de cellules "marquage et balayage", et la gestion d'une mémoire en est améliorée proportionnellement au coût d'une liste libre.

Une telle liste tarife sa liberté: c'est une structure intermédiaire et éphémère construite par le collecteur et détruite par le mutateur. Son origine remonte à LISP 1 et LISP 1.5, où l'on marquait un doublet avec son bit de signe. Définir des marques plus indépendantes permet: (1) de regrouper la phase de démarquage en la situant juste avant la phase de marquage, afin de démarrer avec des doublets uniformément non-marqués, puisque c'est l'unique raison du démarquage; (2) de se libérer de la liste libre en composant le balayage et l'allocation. Cela conduit à un balayage paresseux. Cet article le présente et le justifie, le généralise pour pouvoir libérer de la mémoire à la volée et conclut sur l'efficacité qu'on peut en attendre et sa parallélisation.

Mots-clefs: glanage de cellules, balayage paresseux, LISP, marquage-balayage.

Usual address: LITP-Paris VI (couloir 45-55, 2e étage), 4 place Jussieu, 75252 Paris Cedex 05, FRANCE (uucp: ..!mcvax!inria!litp!od). The stay in Denmark is supported by INRIA.

Introduction

It is particularly striking to see how deep the concept of *free list* is rooted when talking about classical management of storage in LISP systems. The classical survey papers [Cohen 81, 83] show that the need of a free list is taken for granted in non-compacting garbage collection algorithms. Freeing a cons-cell actually consists in chaining it into the so-called free-storage list, in order to reallocate it later on.

One may trace this to the original definition of Garbage Collection [McCarthy 60a, 60b, 62]: when all the available dynamic memory had been allocated, the activity of the system was suspended, active cells were marked and the whole dynamic memory was swept to re-create the free list. This was the way to represent the dynamically available storage.

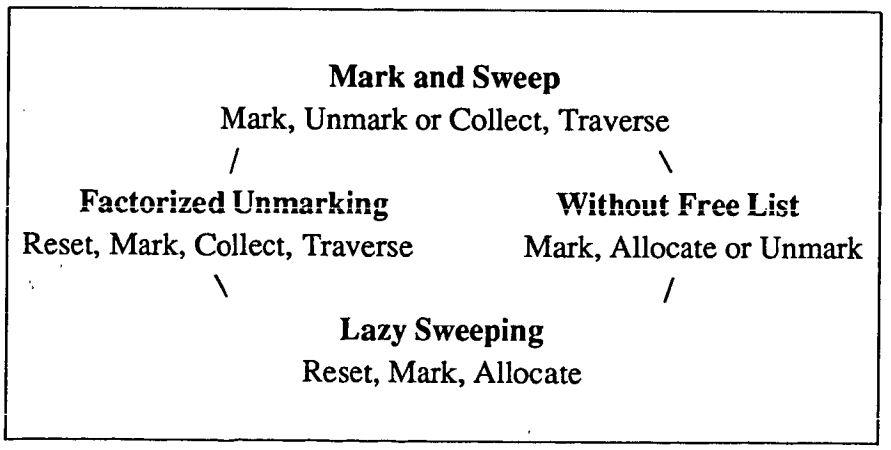
The historical foundation of a free list seems to rely on the way the cons cells were marked: *with their sign bit.* Sweeping thus was absolutely necessary to unmark the active cells, in order to reobtain a consistent state of the system, and avoid misinterpretation of a non-reset sign bit later on. For this reason freed memory cells had to be identifiable by some other means at the end of the collection, hence the free list (constructed during the same bit-resetting sweep) which may be seen as the identification of free cells by location rather than by contents[1].

This one reason has had two major impacts on the mark and sweep algorithm:

— the free list intrinsically exists to identify free storage by location;

— unmarking is integrated in the sweeping process.

Today this economic practice has lost its justification: collectors use more independent marks during the marking phase. Thus there is no longer a strong motivation for sweeping the whole memory. It only serves to construct the free list, which is later destroyed during allocation. The algorithm described below eliminates the need for the free list and thus the need for the sweeping phase of collection. It relies on the following observations:



[1] Note the analogy with *deep binding*: one accesses a binding (resp. a free cell) as an element of an A-list (resp. a constituent of a free list).

A cycle of the original mark and sweep algorithm is composed of the following steps (top part of the figure): (1) active cells are marked by tracing all accessible cells starting from a collection of pointers (registers, stack and global variables); (2) memory is swept, linearly: marked cells are unmarked and unmarked cells are collected and linked in the free list; (3) computation is resumed and cells are allocated from the free list until it is empty, which causes a new collection cycle.

A first remark is that the marking phase only requires the memory to be uniformly unmarked when starting. Unmarking can be factorized just before the marking phase since it does not matter whether a cell is marked or not to use it. The cycle thus becomes (left part of the figure): (1) marks are reset all at once; (2) active cells are marked; (3) memory is swept and unmarked cells are collected in the free list; (4) computation is resumed and the free list is traversed until it is empty.

A second remark conducts to get liberated from the free list: composing sweeping and allocation, there is no more a need for the free list. The cycle becomes (right part of the figure): (1) the active cells are marked at garbage collection time; (2) the memory is traversed at allocation time to allocate unmarked cells and unmark active ones.

These two optimizations can be composed and finally, the cycle becomes (bottom part of the figure): (1) marks are reset; (2) active cells are marked; (3) memory is swept, unmarked cells are allocated and marked cells are left.

This should establish clearly this proposal does improve the good old mark and sweep garbage collector, first in proportion of the free memory (*i. e.* of the free list, which has disappeared) and second in proportion of the non-free memory (*i. e.* of individual unmarking, which can be substituted by a reset of all the marks, according to their representation and location[2]).

1. Mark and Sweep without the Free List

The sweeping phase thus is distributed over the allocation phase[3] and leaves the marking phase invariant: one mark per cell and a recursion stack are needed[4] — the key point being here that the mark of a cell does not interfere with its use.

Grouping the marks in an independent bit-map leads to a number of optimizations: one can coerce the types "bit" and "integer" to divide the time to find a free cell by the current word

---

[2] For example marks can be located in a separate bit map with one bit for each allocatable cell. Alternatively, in the cons-cells of GNU Emacs Lisp, there is one "markbit" per Lisp object, located in the object. The markbit of the CAR is used to mark a cell, and similarly, the markbit of the property list is used to mark a symbol. However, outside of garbage collection, all markbits are always zero. This makes sense, since that garbage collector is mark and sweep (except that it copies strings).

[3] The allocation phase consists in the execution of cell allocation primitives during a computation.

[4] This assumes that a stack is used for marking as suggested in [Knuth 68], but other strategies are possible, such as the use of a second bit [Schorr & Waite 67] or of a bit stack [Wegbreit 72].

length[5] in so far as cells are frequently allocated and abandoned by linear packs (*clusters*) [Clark & Green 77, Clark 79], immediately recognized as null words in the bit map. One may also perform an indirect branch on the bit map to a set of specialized subroutines allocating directly the free cells. For example, the configuration 5 (`101`) would lead to allocate the third and first cells in some page, and the configuration 7 (`1101`) to allocate the fourth, third and first cells. One may even remark that the resulting tree of subroutines which is involved is redundant, as in the example where 5 is a particular case of 7.

These strategies have been implemented in C, after having read [Steele & Sussman 79, 80] and during the realization of a Scheme virtual chip [Danvy 86b]. It has been observed that programs succeeding without any GC take the same time, and that initializations and consumption of cons-cells before the first GC are faster; kamikaze programs which consume and construct more than possibly representable results do obviously not succeed but the fatal error occurs much faster in the one-phase garbage collection: 30% in average; this has been discovered to test exceptional routines which reallocate memory.

It is of course hard to say what an average program is; there are standard tests [Gabriel 85] but they apply for Common Lisp. Our tests have been the first actual programs to run. Self-interpretation of both recursive and continuation-based metacircular definitions [Reynolds 72] has shown an improvement of 15 to 20% in garbage collecting times. The same-fringe problem with flattening method, lazy evaluation or continuation-driven strategies [Hewitt 74] and head abstraction and reduction of Combinatory Logic terms have confirmed that measure.

Finally, one may note that [Hoare 75] names the original McCarthy's mark and sweep garbage collector: "Mark and Scan". This name looks better applied here, concurrently with "Lazy Sweeping".

In the following, the Lazy Sweeping method is justified in part II. It is extended in part III through a voluntary garbage collection. Life without the free list is discussed in part IV. Part V develops issues on efficiency and part VI makes memory management using mark and sweep parallel.

2. Correctness

2.1. Distributed Unmarking

The Lazy Sweeping method is straightforwardly justified by structural induction:

→ by definition, all the active cells are marked and all the free memory is unmarked at the end of the marking phase; thus searching linearly through the memory for the successive free cells, unmarking the marked cells and allocating the unmarked ones is a

[5] One could even pretend that this time ought to be logarithmic rather than linear, given the possibility to test an arbitrary size of memory at a time, although this is not available today.

safe way of allocating;

→ when the whole memory of cells has been traversed, all the cells are homogeneously unmarked; indeed, active cells were marked and have been unmarked and allocated cells already were unmarked;

→ it is sufficient now to mark active cells;

→ once the marking phase is over, the initial state is recovered;

→ marking the statically defined cells while not the others, is a sufficient condition to initialize the system, when bootstrapping it.

2.2. Factorized unmarking

Moving the incremental unmarking from the allocation loop to an isolated extra phase is a formal transformation which does not influence the correctness of the algorithm. To bootstrap the system, it is sufficient to start the linear search *after* the static cells.

3. Voluntary garbage collection

Garbage collection is now reduced to its marking phase and occurs when the whole memory has been searched for a free cell, without success. A *voluntary* GC consists in a collection which explicitly is requested before the system runs out of free cells. The necessary course of action then depends on the unmarking policy chosen.

3.1. Distributed unmarking

To preserve the consistency of the marking semantics, a preliminary operation will occur: the uncovered rest of the memory will be massively declared inactive, *i. e.* unmarked as if it had not been allocated. The marking phase can then start with a homogeneously marked memory.

3.2. Factorized unmarking

In this case, the rest of the memory to be traversed is ignored, all the marks are reset and the marking phase proceeds.

3.3. Mark and sweep

Requiring voluntarily a GC confirms that the free list is a useless construction, since it has been built for nothing. Any special treatment on it (letting its components uncollected by marking them, or others) would involve some new extra overhead, and would upset any compaction phase.

4. Life without the Free List

Two reasons have presided to get liberated from the free list: (1) it was an unessential intermediate result; (2) its management was expensive. As an extra gain, the mark system has been simplified. Now only two primitive operations are needed: to mark a cell and to test whether a cell is marked or not, and furthermore one macro operation to reset all the marks.

But two possibilities disappear with the free list: the free access to an already linked segment of free cells is lost and one can no more free storage on the fly.

Of course it is always possible to manage a personal free list by hand and find in it a source of chained free segments. But this is still a mechanism and not a solution. Let us justify why incrementally freeing memory matters, and investigate its connections with a free list.

4.1. Garbage collection on the fly

4.1.1. Why

It is common for the Lisp implementer to recognize and prevent special cases when some memory is no longer needed, and rather than leaving it to be collected one day, he re-chains it to the free list on the fly.

This point has been shown to be crucial when implementing recursion with the heap (retention) rather than with a vectorial stack (deletion), in order to properly handle captures of continuations [Durieux 81] and more generally snap-back semantics [Reynolds, as quoted in [Jones 85] p 22]. An intelligent marking strategy based on freeing recursive contexts on the fly allows one to come from 80% of CPU time spent in garbage collection[6] to a more reasonable 1 or 2% in the usual cases [Danvy 86a, 87].

This new preoccupation calls again into question the statement from [Baker 78] where freeing storage on the fly is not worth the effort. Nevertheless two SUBRs offer that service in Le_Lisp [Chailloux *et al.* 86], both to free a cons-cell and a list. And a meta-recursive implementation such as Méta-VLisp [Saint-James 87] casually defines the free list as being the current value of a particular symbol.

4.1.2. Why not

In Lazy Sweeping, dynamic memory is conceptually divided in two zones: *before* and *after* the allocation pointer. Therefore:

— either the freed cell follows the allocation pointer, *i. e.* is in the zone to be swept; then since it is an active cell it is marked: freeing reduces to unmarking;

— or the freed cell precedes the allocation pointer, *i. e.* is in the zone which has been swept:

---

[6] Which corroborates the 80% of time spent in garbage collecting in the first Scheme chip [Holloway *et al.* 80], where everything was also represented with cons cells.

→ primarily, the allocation pointer is not supposed to go backwards, but let us extend its semantics, if it is consistent;

→ in the case where unmarking was distributed in the allocation phase, all the cells preceding the allocation pointer are unmarked, as if all of them were free — but this interpretation is inconsistent: the correct one is that this part of memory is ready for the marking phase; thus unless changing the meaning of the mark and recursively adding an extra sweeping phase before marking, there is no way to free a cell using its mark only;

→ the other case is no better: since during allocation the active cells have not been unmarked, nothing can be done: the marks do not reflect any activity any more; allocated cells were marked and any cell may have been abandoned later on.

Thus in the better case a cell can only be voluntarily freed if it is preceded by the allocation pointer, which is not practically tractable. Several possibilities appear: (1) to come back to a hand-managed free list: it would definitely be damaging to any natural coalescence of freed cells[7]; (2) to unmark incrementally non-allocated cells: this gives rise to free cells before and after the allocation pointer, and reverses cyclically the meaning of a mark; (3) to mark incrementally allocated cells: this too reverses cyclically the meaning of a mark; (4) to develop a more sophisticated marking system by zones: indeed, the marking phase still is a bottleneck; (5) to create a new mark per cell: this is developed below.

Here is a simple method to free memory on the fly, since it matters: the idea is to define a second mark per cell and to use it for marking a cell freed on the fly. Defining the second mark in a new table answers to the present requirements (independence + efficiency). When the allocation phase is over, one can start a new allocation phase based on the second mark table: it will allocate what has been freed on the fly during the preceding allocation phase. Of course this is recursively applicable to the first table: at the end of the allocation, the current mark table is systematically reset. In the context of extremely frequent collections on the fly, there will be several allocation cycles between two marking phases and one full sweep before a marking phase. This is definitely realistic when recursion is implemented with the heap [Danvy 87].

The essence of this problem is that it is garbage rather than active cells which are considered, both after the marking phase and when collected on the fly. The free list offered a convenient hook.

4.2. Garbage collection of data with fixed format

The cons cells from Lisp provide a classical example of data with fixed format. From a general point of view, the allocation of these data which consists in taking the first one from some free list can be safely replaced by the criterion permitting to put that data in the

[7] For example in the case of contiguous clusters freed. That property naturally belongs to the Stop and Copy garbage collector, where there is no free list either, but only half the memory space.

free list — provided that the marking system does not interfere with anything else.

As an interesting case, let us consider the symbols in a Lisp system. They have a fixed format and generally are stored in a free list. To garbage collect them is not trivial, since they are usually involved in some kinds of hash-code (on the Print-Name), for efficiency. Still, it is possible to get liberated from their free list, with the following idea: after some marking strategy permitting to determine which symbol is free or active, the sweeping phase also disappears, and inactive symbols are *not* removed from their hash-code class. On the contrary, when a new symbol is to be created, it is sufficient to use an unmarked (and thus free) entry in its hash-code class and re-use it. Furthermore, one can collect its Print-Name on the fly or better re-use it (if that string is free), as a natural instance of the hash-code class.

4.3. Garbage collection of data with variable format

That case is more specific since the criterion of *coalescence* intervenes, as a way to avoid memory fragmentation. It is classically accomplished with an extra phase.

5. Issues on efficiency

We have completely implemented lazy sweeping in C, including the possible reversions of marks, in the fall of 1986 at DIKU. The results have been an average improvement of 20% in computation times.

However the main optimization reveals to be factorized rather than distributed unmarking[8]. Indeed, as pointed out by Kent Dybvig [Dybvig 87], building the free list is faster than the allocation of free cells in lazy sweeping, because one traverses the memory in one loop and the other traverses the memory in lots of little loops, repeating the loop prelude and postlude in proportion to the number of free cells. On the other hand, with a free list, the allocation time which is also proportional to the number of free cells should be added. From the strict point of view of efficiency, such considerations conduct to simply count the number of elementary instructions in the collection and allocation and to weight it with the number of freed cells[9]. From a more conceptual point of view, this leads to define the consumer and the allocator as two coroutines [Conway 63]: the consumer is the mutator [Dijkstra *et al.* 78]; the allocator is the sweeping phase and as a coroutine is a perpetual loop with one prelude and one postlude. This eliminates the overhead of the little allocation loops between two marking phases.

Mark and sweep has been defined procedurally with a free list: allocating was performed with a procedural call. Getting liberated from the free list conducts to two coroutines for consuming and allocating. Surprisingly, considering the collector and the mutator as two independent processors running in parallel makes the free list reappear: because it

[8] Peter Deutsch [Deutsch 87] told us that unmarking was factorized before the marking phase, in all the mark and sweep garbage collectors he had implemented but that they all had a free list.

[9] Obtained by decrementing a counter in the marking phase, each time a new cell is marked.

represents then all the free cells to be allocated later on. And when the collector has finished to sweep the memory, it can restart a marking phase in advance, breaking the mark and sweep bottleneck.

6. Towards a parallel mark and sweep

If the marking is to be performed in parallel with the computation, it may not have detected all cells not active at the end of the marking phase. This is still a safe method, as the rest will be collected in the next cycle. The important point is not that when the marking phase ends, there are too many marked cells, but that all cells which are active are marked at the end of the marking phase. This is done by extending the set of original pointers (registers, stack, global variables) to all the cells allocated, assigned or side-effected meanwhile (this includes the free list), with an appropriate synchronization: a newly considered cell which is not marked is pushed on the marking stack[10]. If the marking phase terminates without the free list being exhausted, there is no interruption and the sweeping phase can start. If the free list is exhausted during a marking phase, then either allocation becomes lazy [Friedman & Wise 76] for some time, or more realistically the mutator is interrupted and helps the collector to finish in parallel the marking phase, with the very strong reason that they do not need any synchronization [Lang & Dupont 87].

This illustrates simply the possible multiplication of processors devoted to garbage collection (and their cooperation to perform the main marking phase):

— either on different parts of the memory [Lang & Dupont 87];

— or on the same parts of memory, but with independent marks: the idea here would be to have, at each moment, different viewpoints of a same configuration and to be able to select one of them at any moment; the use would be for a distributed garbage collector: changing of viewpoint would occur for example when a task ends; all the memory that it used would become instantly free, because it *already* is free in the new viewpoint.

Conclusion and perspectives

This article describes how to get liberated from the free list, by transforming the mark and sweep garbage collection and its associated allocation to lazy sweeping. This transformation is merely a composition and simplification of redundant and/or expensive operations: (1) the free list itself as an useless intermediary result; (2) unmarking as an iterative access to one unique mark. All of this has been presented as a consequence of making marks independent from the use of the markable objects and identifying independent operations to simplify their composition.

---

[10] This can be realized without any test by redefining periodically the elementary routines accessing to cells.

Lazy Sweeping does not interfere with all the possible improvements in the marking phase: unfolding the current unmarked cell rather than pushing blindly its CDR, compacting the marking stack when it overflows and resume marking, and so on. A parallel mark and sweep garbage collector has even been sketched.

The modified algorithm still requires one (independent!) mark per cell and a stack for recursively marking linked nets. It may be extended to compacting garbage collection with an extra phase. It is presently used in a new Lisp system: LIPSC, that we have developed on the Intel's Hypercube[11] where it corresponds to a need: there is only 256K available per node and semi-spaces are out of the question.

Acknowledgements

Jean-Louis Durieux one day suggested to distribute the sweeping phase, leading to a *balayage paresseux*, independently named *lazy sweeping* by Mads Rosendahl, later on. Patrick Greussay guided us through the bibliography of LISP. Neil D. Jones has recalled us (smiling) the hack of indirect calls through the bit map. Emmanuel Saint-James, Francis Kloss and the META working group have received the first draft of this. Peter Sestoft, Torben Mogensen, Karoline Malmkjær, Kristoffer Høgsbro Holm, Bernard Lang, Francis Dupont and Jérôme Chailloux have re-read and commented it. Thanks to Christopher T. Haynes and R. Kent Dybvig for their very lucid discussion and analysis of lazy sweeping, and to Matthias Felleisen for his challenging question about a parallel mark and sweep garbage collector. Finally, thanks to David K. Gifford and Pierre Jouvelot for their invitation to lecture this at the MIT Laboratory for Computer Science in June 1987.

Translation

Here are some proposals to translate "Garbage Collection" in French: *Glanage de Cellules* of course[12], but it could be *Gabelle Chronique*, or even, with a free list: *Glanage Central* or *Gaulage Cataleptique*, since everything is suspended during garbage collection. An incremental garbage collector could be a *Grappillage Continu*, or *Conjoint*, while a two semi-space one would be a *Glanage Colporteur*. If it is a compacting one, then it would be a *Glanage Compresseur* or better *Colmateur*. Finally reference counting could be named *Glanage au Compte-goutte*.

References

Henry G. Baker Jr.:
LISP processing in real-time on a serial computer, CACM Vol 21, No 4 pp 280-294 (April 1978)

Jérôme Chailloux, Matthieu Devin, Francis Dupont, Jean-Marie Hullot, Bernard Serpette, Jean Vuillemin:

[11] Alias iPSC. Intel and iPSC are registered trademarks from Intel Corporation.

[12] The epidermic connotation of "*Grattage de Cellules*" has been fatal to it.

Le_Lisp Version 15.2, le Manuel de Référence, 2e édition, rapport INRIA, Domaine de Voluceau, Rocquencourt, France (May 1986)

Douglas W. Clark, C. Cordell Green:

An empirical study of list structure in LISP, CACM Vol 20, No 2 pp 78-87 (February 1977)

Douglas W. Clark:

Measurements of dynamic list structure use in Lisp, IEEE Transactions on Software Engineering, Vol. SE-5, No 1 pp 51-59 (January 1979)

Jacques Cohen:

[Cohen 81] *Garbage Collection of Linked Data Structures*, ACM Computer Surveys, Vol. 13, No 3 pp 341-367 (September 1981)

[Cohen 83] *Comparison of Compacting Algorithms for Garbage Collection*, ACM Transactions on Programming Languages and Systems, Vol 5, No 4 pp 532-553 (October 1983)

M. E. Conway:

Design of a separable transition-diagram compiler, CACM Vol. 6 pp 396-408 (July 1963)

Olivier Danvy:

[Danvy 86a] *Machines virtuelles pour Langages Applicatifs et Langages Acteur: réalisations et programmation*, PH. D, Université Paris VI, LITP 86-57, Paris (June 1986)

[Danvy 86b] *From Meta-Circularity to Functional Architectures: Micro-VLISP, its definition, simulation and threading*, Datalogisk Kollokvium, Copenhagen University, Denmark (November 7th, 1986)

[Danvy 87] *Memory Allocation and Higher-Order Functions*, DIKU report No 87/1, also in the Proceedings of the SIGPLAN '87 Symposium on *Interpreters and Interpretive Techniques* pp 241-252, Saint-Paul, Minnesota (June 24-26, 1987)

L. Peter Deutsch:

Personal communication, Saint-Paul, Minnesota (June 26th, 1987)

E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Sholten, E. F. M. Steffens:

On-the-fly Garbage Collection: an Exercise in Cooperation, CACM Vol. 21, No 11 pp 996-975 (November 1978)

Jean-Louis Durieux:

Sémantique des liaisons nom-valeur: application à l'implémentation des λ-langages, thèse d'état, Université Paul Sabatier, Toulouse (September 1981)

R. Kent Dybvig:

Personal communication, Indiana University, Bloomington (June 18th, 1987)

Dan Friedman, David Wise:

Cons should not evaluate its arguments, in *Automata, Languages and Programming*, S. Michaelson and R. Milner (eds) pp 257-284, Edinburgh University Press, Edinburgh (1976)

Richard P. Gabriel:

Performance and Evaluation of Lisp Systems, The MIT Press (1985)

Carl E. Hewitt, Peter Bishop, Richard Steiger, Irene Greif, Brian Smith, Todd Matson, Roger Hale:

Behavioral semantics of non-recursive control structures, Proceedings Colloque sur la Programmation, B. Robinet (ed), Springer Verlag, Lecture Notes in Computer Science Vol. 19 pp 385-407, Paris (April 1974)

C. A. R. Hoare:

Recursive Data Structures, International Journal of Computer and Information Science, Vol 4, No 2 pp 105-132 (1975)

Jack Holloway, Guy L. Steele Jr., Gerald Jay Sussman, Alan Bell:

The SCHEME-79 Chip, MIT AI Lab., AI Memo No 559, Cambridge, Massachusetts (January 1980)

Neil D. Jones:

Towards automating the transformation of programming language specifications into compilers, DIKU report 85/9, University of Copenhagen, Denmark (1985)

Donald E. Knuth:

The Art of Computer Programming, Vol. 1, Addison-Wesley (1968)

Bernard Lang, Francis Dupont:

Incremental Incrementally Compacting Garbage Collection, Proceedings of the SIGPLAN '87 Symposium on *Interpreters and Interpretive Techniques* pp 253-263, Saint-Paul, Minnesota (June

24-26, 1987)

John McCarthy *et al.*:

[McCarthy 60a] *LISP 1 programmer's manual*, Computation Center and Research Laboratory of Electronics, MIT, Cambridge, Mass. (March 1960)

[McCarthy 60b] *Recursive functions of symbolic expressions and their computation by machine, part I*, CACM, Vol. 3, No 3 pp 184-195 (March 1960)

[McCarthy 62] *LISP 1.5 programmer's manual*, MIT Press, Cambridge, Mass. (1962)

John Reynolds:

Definitional Interpreters for higher-order programming languages, Proceedings 25th ACM National Conference pp 717-740, New York (1972)

Emmanuel Saint-James:

META-VLISP Version 1, Manuel d'Utilisation de l'Interprète, LITP/CNDP report, Paris (May 1987)

H. Schorr, W. M. Waite:

An efficient machine-independent procedure for Garbage Collection in various list structures, CACM Vol 10, No 8 pp 501-506 (August 1967)

Guy L. Steele Jr., Gerald Jay Sussman:

Design of LISP-Based Processors, MIT AI Lab., AI Memo No 514, Cambridge, Massachusetts (March 1979), also in CACM Vol. 23, No 11 pp 628-645 (November 1980)

Ben Wegbreit:

A space-efficient list structure tracing algorithm, IEEE Trans. Comp. Vol. 21, No 9 pp 1009-1010 (September 1972)

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique